Technische Universität Dresden
Institut für Theoretische Informatik
Dr. Monika Sturm

Wintersemester 09/10

Funktionale Programmierung und Typtheorie

6. Übungsblatt

Aufgabe 1

Das Konzept der Monaden kann u. a. dazu verwendet werden, um in reinen funktionalen Programmiersprachen Möglichkeiten zur Generierung von Ausgaben zu schaffen. Die zu diesem Zweck benötigte Ausgabe-Monade repräsentiert eine Berechnung, die als „Seiteneffekt" Ausgaben erzeugt. Als zugrunde liegende Datenstruktur wird dabei ein Paar verwendet, das aus Ausgabe und dem eigentlichem Wert besteht.

Untersuchen Sie den folgenden lückenhaft gegebenen Interpreter auf der Basis einer Ausgabe-Monade hinsichtlich seiner Funktionsweise und fügen Sie die fehlenden mit ... gekennzeichneten Teile ein. Testen Sie anhand einiger einfacher Beispiele seine Funktionsweise.

```
-- Monade fuer Ausgabeoperationen

type Output= String

type M a = (Output, a)

unitM :: a -> M a
unitM a = ("", a)

bindM :: M a -> (a -> M b) -> M b
m `bindM` k = let (r, a) = m
                  ...    = ...
              in (r++s, b)

showM :: M Value -> Output
showM (s, a) = "Output: " ++ s ++ " Value: " ++ showval a

outM :: Value -> M ()
outM a = (showval a ++ "; ", ())


-- call-by-value-Interpreter

type Name = String

data Term = Var Name
```

```
          | Con Int
          | Add Term Term
          | Lam Name Term
          | App Term Term
          | Out Term
data Value = Wrong
           | Num Int
           | Fun (Value -> M Value)

type Environment = [(Name, Value)]

showval :: Value -> String
showval Wrong   = "<wrong>"
showval (Num i) = showInt i ""
showval (Fun f) = "<function>"

interp :: Term -> Environment -> M Value
interp (Var x) e   = fetch x e
interp (Con i) e   = unitM (Num i)
interp (Add u v) e = ...
                     add a b))
interp (Lam x v) e = unitM (Fun (\a -> interp v ((x, a):e)))
interp (App t u) e = ...
                     apply f a))
interp (Out u) e   = interp u e `bindM` (\a ->
                     outM a `bindM` (\() ->
                     unitM a))

fetch :: Name -> Environment -> M Value
fetch x []        = ...
fetch x ((y,b):e) = if x==y then unitM b else fetch x e

add :: Value -> Value -> M Value
add (Num i) (Num j) = ...
add a b             = unitM Wrong

apply :: Value -> Value -> M Value
apply (Fun k) a = k a
apply f a = unitM Wrong

-- Aufruf des Interpreters
test :: Term -> String
test t = showM (interp t [])
```

Aufgabe 2

Definieren Sie mithilfe der Zustandsmonade eine Funktion

`flatten :: Tree a -> [a]`,

die alle Baumeinträge in einer Liste aufsammelt.

Aufgabe 3

Nicht-deterministische Berechnungen lassen sich mit einer geeigneten Monade ausdrücken. Die im Kapitel 5 der Vorlesung vorgestellten Parser erlauben eine nicht-deterministische Auswahl mithilfe einer Liste der möglichen Ergebnisse.

Definieren Sie für eine Anwendung in diesem Kontext eine Listenmonade `L a`.